# *Divide and Conquer*

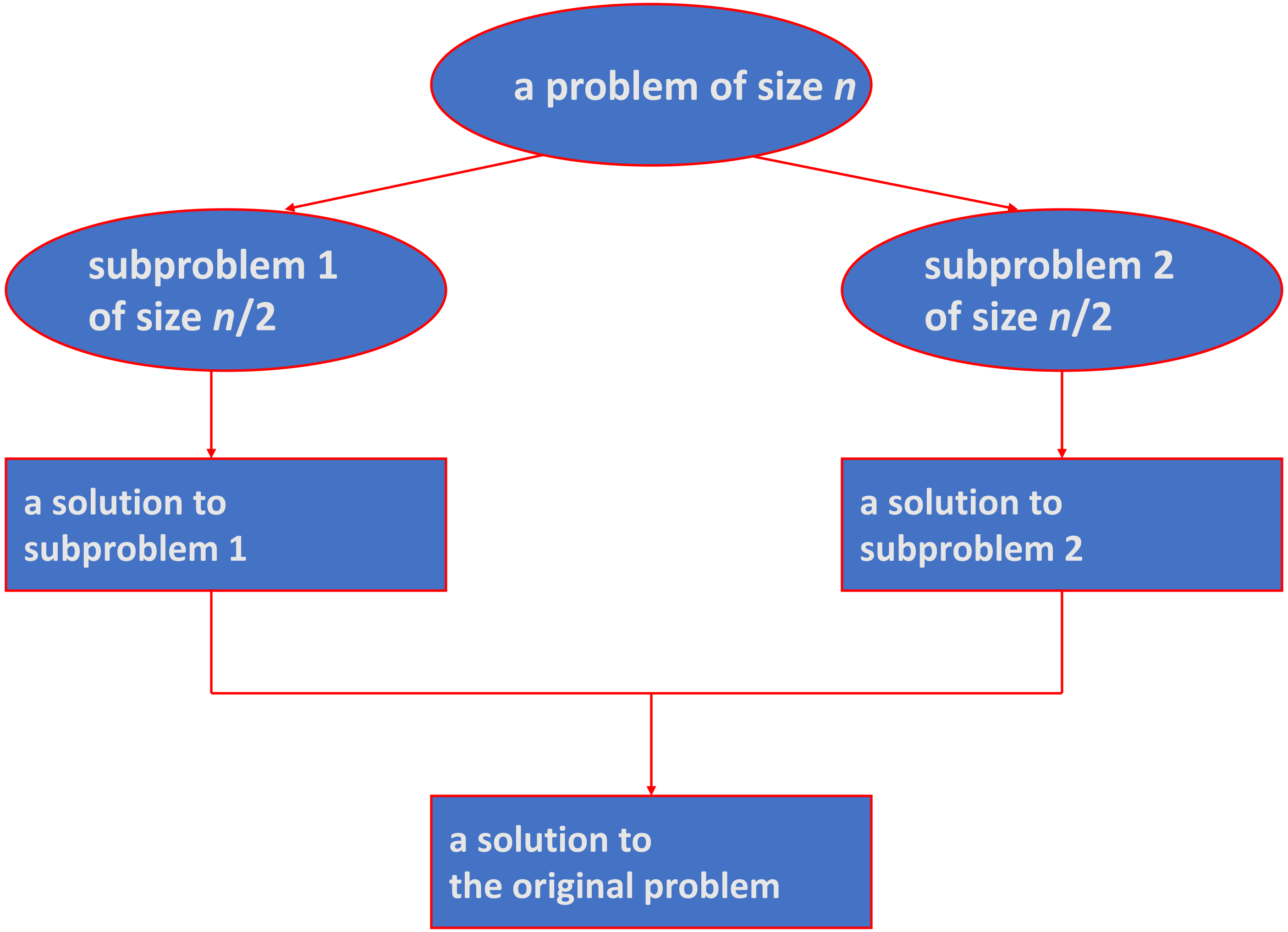
# *Divide-and-Conquer*

- The ancient Roman politicians understood an important principle of good algorithm design.
  - **divide your enemies** (by getting them to distrust each other)
  - **then conquer them piece by piece**.
- In algorithm design, the idea is :
  - to take a problem on a large input, break the input into smaller pieces,
  - solve the problem on each of the small pieces,
  - then combine the piecewise solutions into a global solution.

# *Divide and Conquer*

- The main elements to a divide-and-conquer solution are :
  - **Divide** the problem into a small number of pieces,
  - **Conquer :** solve each piece, by applying divide-and-conquer recursively to it,
  - **Combine** the pieces together into a global solution

# *Divide-and-Conquer*



# *Merging two sorted arrays*

- Combining two ordered arrays :

A

| | | | |
|---|---|---|---|
| 3 | 7 | 15 | 21 |

B

| | | | | |
|---|---|---|---|---|
| 2 | 5 | 6 | 8 | 14 |

C

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 5 | 6 | 7 | 8 | 14 | 15 | 21 |

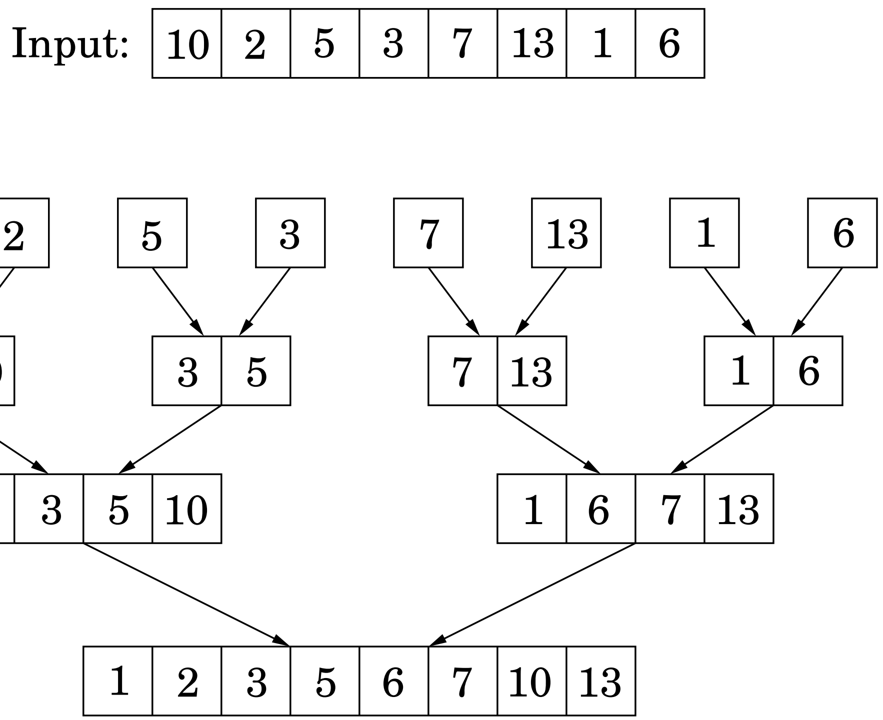
# *Merging two sorted arrays*

```
void merge(int A[], int B[], int C[] int nA, int nB)
{
	int iA=0, iB=0, iC=0;
	while((iA < nA) && (iB < nB))
	{
			if(A[iA] < B[iB])
					C[iC++] = A[iA++];
			else
					C[iC++] = B[iB++];
	}
	while(iA < nA) C[iC++] = A[iA++];
	 while(iB < nB) C[iC++] = B[iB++];
}
```

Complexity : O(N)

# Merge Sort *(by John von Neuman in 1945)*

**Divide :** Split A down the middle  into two subsequences
**Conquer :** Sort each subsequence recursively
**Combine :** Merge the two sorted subsequences into a single sorted list

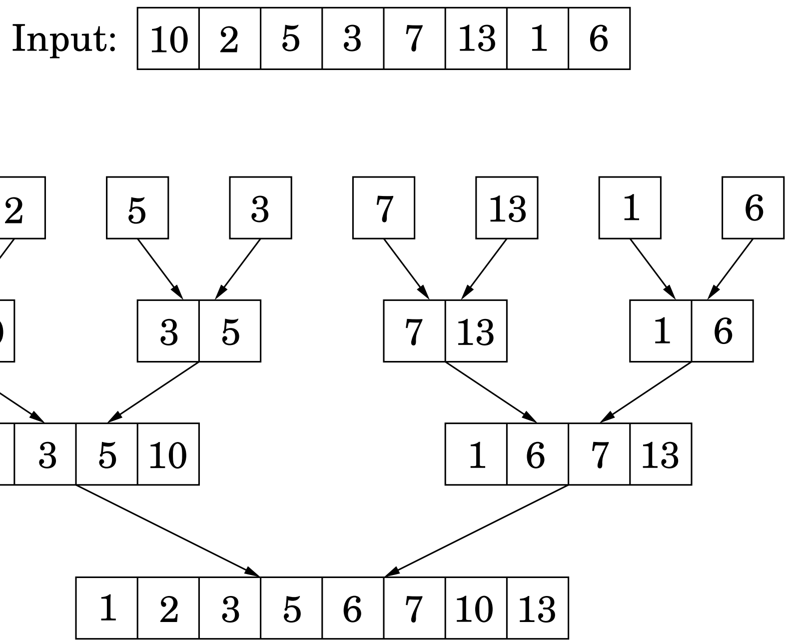


# Merge Sort

```
void mergeSort(int A[], int p,int r)          // sort sub array A[p..r]
{
     int q;
     if(p < r)                                // we have at least two items
     {
          q = (p + r ) / 2;
          mergeSort(A, p , q);                // sort A[p..q]
          mergeSort(A, q+1,r);                // sort A[q+1..r]
          merge(A,p,q,r);
     }
}
```

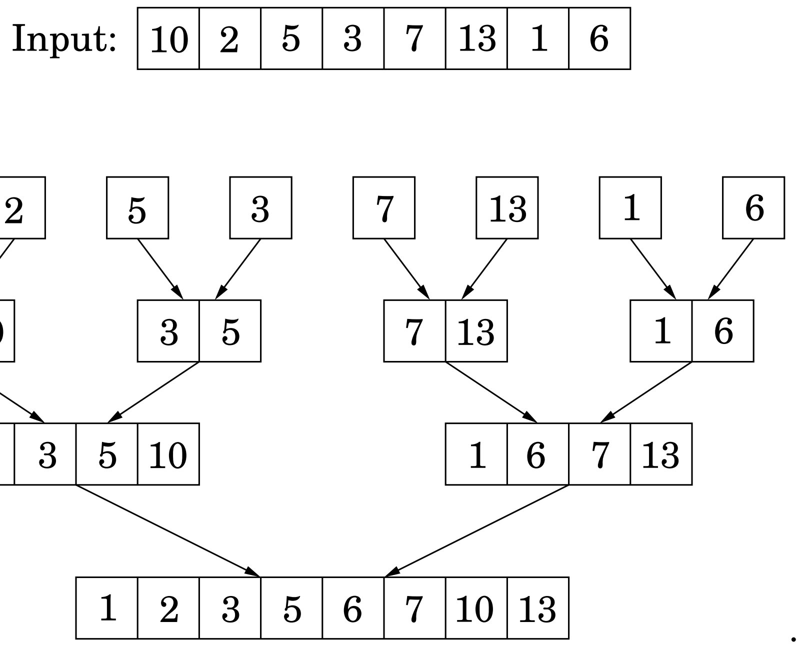
# Merge Sort

```
void mergeSort(int A[], int p,int r)
{
	int q;
	if(p < r)
	{
		q = (p + r ) / 2;
		mergeSort(A, p , q);
		mergeSort(A, q+1,r);
		merge(A,p,q,r);
	}
}
```



# *Merge Sort*

A[0..7]
MergeSort(A,0,7)
    MergeSort(A,0,3)
        MergeSort(A,0,1)
            MergeSort(A,0,0)
            MergeSort(A,1,1)
            **Merge(A,0,0,1)**
        MergeSort(A,2,3);
            MergeSort(A,2,2)
            MergeSort(A,3,3)
            **Merge(A,2,3)**
        **Merge(A,0,1,3)**
    MergeSort(A,4,7)
    **.........**
    **Merge(A,0,3,7)**



# Merge Sort

```
void merge(int A[], int p, int q, int r)
{
         int i, j, k, nB, *B;
        nB = r – p + 1;
        B = (int *) malloc (nB * sizeof(int));
        i = p ;  j = q + 1;  k= 0;
        while((i <= q) && (j <=r))
        {
                if(A[i] < A[j])
                        B[k++] = A[i++];        // copy from left subArray
                else
                        B[k++] = A[j++];        // copy from right subArray
        }
        while(i <= q)
                B[k++] = A[i++];
         while(j <= r)
                B[k++] = A[j++];
        copy(A,B);
        free(B);
}
```

# Efficiency of Merge Sort

**C(N) :** the number of compares to sort an array of length N

$C(0) = C(1) = 0$

$C(N) = C(N/2) + C(N/2) + N$ for $N > 0$

$C(N) = 2 * C(N/2) + N$

$C(N/2) = 2 * C(N/4) + N/2$

$C(N) = 2 * (2 * C(N/4) + N/2) + N$

$= 4 * C(N/4) + 2*N$

$= 4 * (2* C(N/8) * N/4) + 2*N$

$= 8 * C(N/8) + 3*N$

if $N = 2^i$

$C(N) = 2^i * C(2^i / 2^i) + i*N$

$i = lg_2 N$

$= N * C(1) + lg_2 N*N$

$= N + lg_2 N*N$

**Merge Sort Complexity : $O(N*lg_2 N)$**

# Merge Sort

Running time estimates:

- Home PC executes $10^8$ compares/second.
- Supercomputer executes $10^{12}$ compares/second.

| | insertion sort ($N^2$) | | | mergesort (N log N) | | |
|---|---|---|---|---|---|---|
| computer | thousand | million | billion | thousand | million | billion |
| home | instant | 2.8 hours | 317 years | instant | 1 second | 18 min |
| super | instant | 1 second | 1 week | instant | instant | instant |

# Merge Sort

- Primary drawback : it requires extra space
- Use insertion sort for small arrays (length 15 or less)
- Before merge, test whether the array is already sorted ( a[mid] < a[mid+1] )

Running time estimates:

- Home PC executes $10^8$ compares/second.
- Supercomputer executes $10^{12}$ compares/second.

| | insertion sort ($N^2$) | | | mergesort (N log N) | | |
|---|---|---|---|---|---|---|
| computer | thousand | million | billion | thousand | million | billion |
| home | instant | 2.8 hours | 317 years | instant | 1 second | 18 min |
| super | instant | 1 second | 1 week | instant | instant | instant |

# Quick Sort (by Sir Charles Antony Richard Hoare in 1960)

- Shuffle the array
- Partition the array for pivot X
  - No larger element to the left of X
  - No smaller element to the right of X
- Sort each piece recursively

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| input | Q | U | I | C | K | S | O | R | T | E | X | A | M | P | L | E |
| shuffle | K | R | A | T | E | L | E | P | U | I | M | Q | C | X | O | S |
| partition | E | C | A | I | E | K | L | P | U | T | M | Q | R | X | O | S |
| sort left | A | C | E | E | I | K | L | P | U | T | M | Q | R | X | O | S |
| sort right | A | C | E | E | I | K | L | M | O | P | Q | R | S | T | U | X |
| result | A | C | E | E | I | K | L | M | O | P | Q | R | S | T | U | X |

*partitioning item*

*not greater*

*not less*

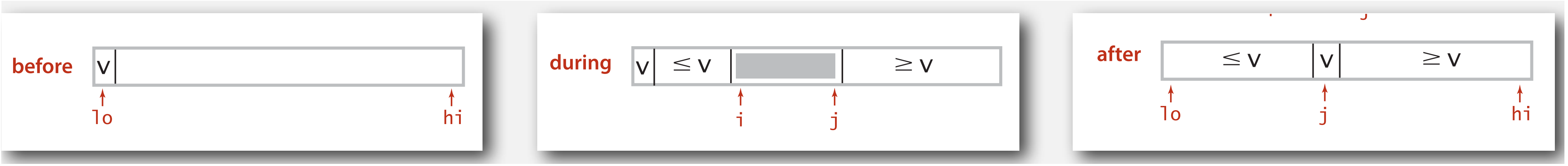
# Quick  Sort Partitioning

- Scan i from left for an item that belongs on the right.
- Scan j from right for an item that belongs on the left.
- Exchange a[i] and a[j].
- Repeat until pointers cross

| | | | v | | | | | | | a[i] | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | i | j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| initial values | 0 | 16 | K | R | A | T | E | L | E | P | U | I | M | Q | C | X | O | S |
| scan left, scan right | 1 | 12 | K | R | A | T | E | L | E | P | U | I | M | Q | C | X | O | S |
| exchange | 1 | 12 | K | C | A | T | E | L | E | P | U | I | M | Q | R | X | O | S |

# *Quick Sort Partitioning*

- Scan **i** from left for an item that belongs on the right.
- Scan **j** from right for an item that belongs on the left.
- Exchange **a[i]** and **a[j]**.
- Repeat until pointers cross



# Quick Sort Partitioning

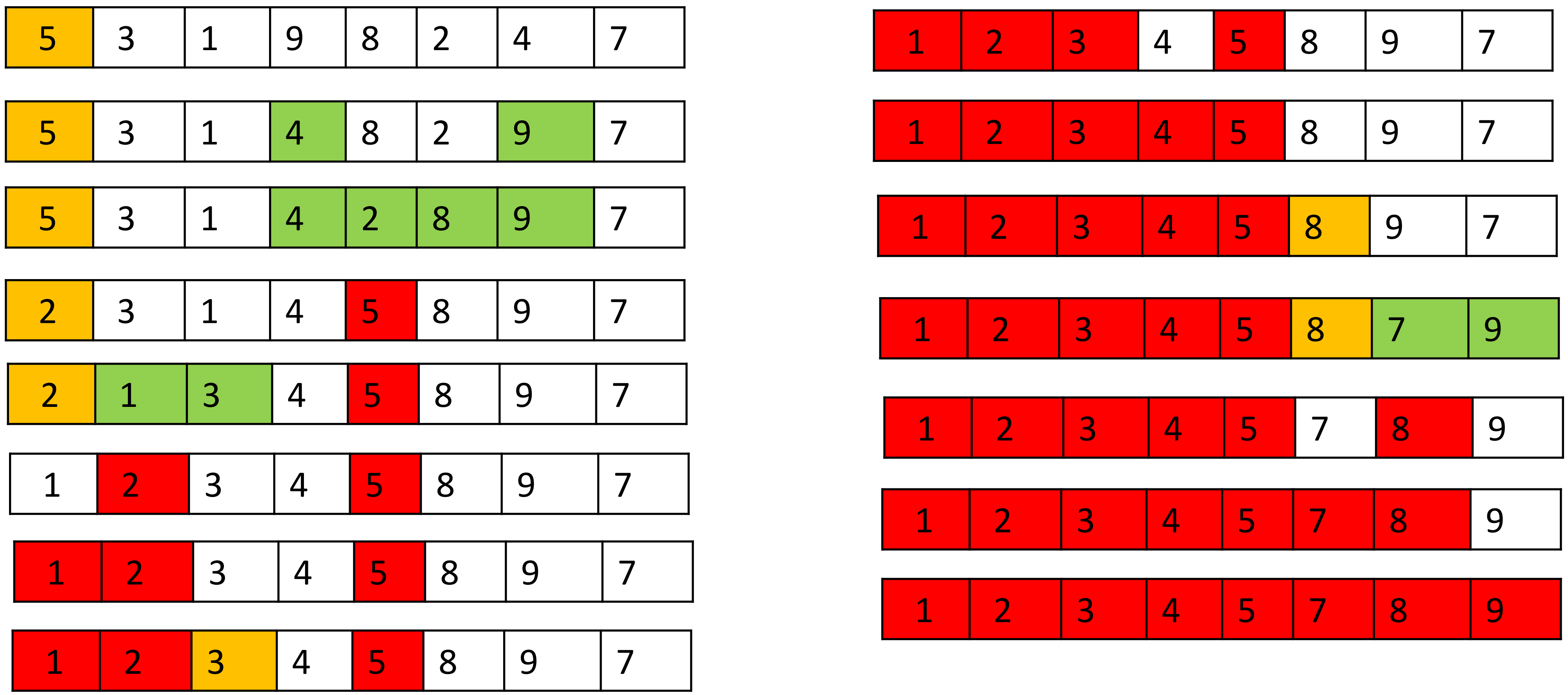
| | | | v | | | | | | | a[i] | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | i | j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| initial values | 0 | 16 | K | R | A | T | E | L | E | P | U | I | M | Q | C | X | O | S |
| scan left, scan right | 1 | 12 | K | R | A | T | E | L | E | P | U | I | M | Q | C | X | O | S |
| exchange | 1 | 12 | K | C | A | T | E | L | E | P | U | I | M | Q | R | X | O | S |
| scan left, scan right | 3 | 9 | K | C | A | T | E | L | E | P | U | I | M | Q | R | X | O | S |
| exchange | 3 | 9 | K | C | A | I | E | L | E | P | U | T | M | Q | R | X | O | S |
| scan left, scan right | 5 | 6 | K | C | A | I | E | L | E | P | U | T | M | Q | R | X | O | S |
| exchange | 5 | 6 | K | C | A | I | E | E | L | P | U | T | M | Q | R | X | O | S |
| scan left, scan right | 6 | 5 | K | C | A | I | E | E | L | P | U | T | M | Q | R | X | O | S |
| final exchange | 6 | 5 | E | C | A | I | E | K | L | P | U | T | M | Q | R | X | O | S |
| result | 6 | 5 | E | C | A | I | E | K | L | P | U | T | M | Q | R | X | O | S |

Partitioning trace (array contents before and after each exchange)

# *Quick Sort*

```
void quicksort(int a[], int l, int r)
{
        if(l<r)
        {
                adr = partition(a,l,r);
                quickSort(a,l,adr-1);
                quickSort(a,adr+1,r);
        }
}
```

# Example



5 3 1 9 8 2 4 7
5 3 1 4 8 2 9 7
5 3 1 4 2 8 9 7
2 3 1 4 5 8 9 7
2 1 3 4 5 8 9 7
1 2 3 4 5 8 9 7
1 2 3 4 5 8 9 7
1 2 3 4 5 8 9 7
1 2 3 4 5 8 9 7
1 2 3 4 5 8 9 7
1 2 3 4 5 8 9 7
1 2 3 4 5 8 7 9
1 2 3 4 5 7 8 9
1 2 3 4 5 7 8 9
1 2 3 4 5 7 8 9

# *Efficiency of Quick Sort*

Running time estimates:

- Home PC executes $10^8$ compares/second.
- Supercomputer executes $10^{12}$ compares/second.

| | insertion sort ($N^2$) | | | mergesort (N log N) | | | quicksort (N log N) | | |
|---|---|---|---|---|---|---|---|---|---|
| computer | thousand | million | billion | thousand | million | billion | thousand | million | billion |
| home | instant | 2.8 hours | 317 years | instant | 1 second | 18 min | instant | 0.6 sec | 12 min |
| super | instant | 1 second | 1 week | instant | instant | instant | instant | instant | instant |

# *Efficiency of Quick Sort*

- Bestcase: (split in the middle) O (nlogn)
- Worst case: (sorted array) O( $n^2$ )
- Average case: random arrays—Θ(nlogn)
- Improvements:
  - better pivot selection: median of three partitioning avoids worst
- switch to insertion sort on small subfiles
- Quicksort is not stable

# Sorting Summary

| | inplace? | stable? | worst | average | best | remarks |
|---|---|---|---|---|---|---|
| selection | ✔ | | $N^2 / 2$ | $N^2 / 2$ | $N^2 / 2$ | *N* exchanges |
| insertion | ✔ | ✔ | $N^2 / 2$ | $N^2 / 4$ | N | use for small *N* or partially ordered |
| shell | ✔ | | ? | ? | N | tight code, subquadratic |
| merge | | ✔ | N lg N | N lg N | N lg N | *N* log *N* guarantee, stable |
| quick | ✔ | | $N^2 / 2$ | 2 N ln N | N lg N | *N* log *N* probabilistic guarantee<br>fastest in practice |